An Introduction to Time-Constrained Automata

Matthieu Lemerre, Vincent David and Christophe Aussaguès
CEA, LIST, Embedded Real-Time System Lab
91191 Gif-sur-Yvette Cedex, France
Firstname.Lastname@cea.fr

Guy Vidal-Naquet
SUPELEC
91192 Gif-sur-Yvette Cedex, France
Guy.Vidal-Naquet@supelec.fr

We present time-constrained automata (TCA), a model for hard real-time computation in which agents behaviors are modeled by automata and constrained by time intervals.

TCA actions can have multiple start time and deadlines, can be aperiodic, and are selected dynamically following a graph, the time-constrained automaton. This allows expressing much more precise time constraints than classical periodic or sporadic model, while preserving the ease of scheduling and analysis.

We provide some properties of this model as well as their scheduling semantics. We show that TCA can be automatically derived from source-code, and optimally scheduled on single processors using a variant of EDF. We explain how time constraints can be used to guarantee communication determinism by construction, and to study when possible agent interactions happen.

1 Introduction

Most concrete implementations of real-time systems use only two different kinds of tasks: *periodic tasks* and *sporadic tasks*. Periodic tasks must execute one job per period of time, and are meant for regular processing. Sporadic tasks are meant for processing of events of limited occurrence, with jobs having a deadline relative to the arrival of the event. We believe that these kinds of tasks are not expressive enough in many situations. Restricting real-time systems to use only them puts heavy constraints on the design of real-time applications, which make them harder to design, implement and analyze.

For instance, the timing behavior of complex tasks can be specified using timed automaton[1], whose behavior is not cyclic. These kind of tasks fit neither the periodic nor the sporadic task model, making the translation to these tasks inefficient. Common example of complex timing behaviors are degraded mode, multi-phase applications (e.g. take-off/flight/landing phases of air travel...). An issue that motivates the need for more accurate task models is *jitter*, which is the variation between successive executions of a periodic task. Current real-time methodologies consist of analyzing jitter once the design is done and the execution times of the tasks are known. Thus, the whole design has to be modified if the jitter of a task is too high. It is also impossible to take into account the fact that different tasks have different degrees of sensitivity to jitter. On the contrary, the time-constrained task model we present allows to express the maximum jitter directly in the model, making it a constraint that the scheduling algorithm has to enforce. Moreover the maximum jitter can appear in the specification, design and code of each task, and bounding of jitter is thus guaranteed by construction.

We claim that using a more expressive task model allows an easier development (less transformations need to be done from specification to the implementation) and better verification, thus increasing the safety of the system. This is why OASIS [2], a toolchain implementing a subset of the time-constrained task model, is used in hard real-time safety-critical environments, such as the nuclear industry [3]. This toolchain comprises in particular a specific compiler, a microkernel and operating system services, whose behaviors are functionally described in this article. We also claim that using this model allows to reduce

Bliudze, Bruni, Grohmann, Silva (Eds.):
Third Interaction and Concurrency Experience (ICE 2010)
EPTCS 38, 2010, pp. 83–98, doi:10.4204/EPTCS.38.9

hardware costs, because accurate modeling of task needs and exact feasibility analysis allows to achieve a very high utilization, on single and multiple processors.

The purpose of this paper is to present formal semantics of the time-constrained automata model, and proof of some important results (optimality of EDF, communication determinism) that come from using this model. The paper is structured as follows: Section 2 present related work, and Section 3 the time-constrained task model. Section 4 provides example uses and shows how they can be derived automatically from source code expressed in a suitable language. Section 5 shows how time-constrained tasks can be scheduled, and gives an optimal scheduling algorithm on single processors based on EDF.

2 Related work

Timed automata Time-constrained automata is a model to be used for scheduling rather than for accurate model-checking. Notable restrictions from model-checking models such as timed automata [1] is that we abstract the control flow logic by considering that the choice between several transitions is nondeterministic. Moreover, it is the choice of a transition that acts on a time constraint, rather than having time constraints acts on the control flow logic.

However, time-constrained automata can be modeled using timed automata, using only two clocks. It is thus a simpler model, i.e. less expressive, but easier to analyze, than the general timed automata.

Task models The main characteristics of our model is that it allows to express infinite computations and allows dynamic modification of time constraints. Usual task models perform dynamic release of fixed jobs (i.e. the start time, deadline of the execution time does not evolve) [4]. By contrast, a time-constrained automaton can be viewed as one job that changes dynamically.

Moreover, timing behavior can change depending on choices not expressed in the automaton: so the task describes a set of possible sets of timing requirements, rather than a fixed set of timing requirements.

There have been other attempts to provide more accurate real-time models with multiple deadlines and start times [5], but they are event-triggered and impose cyclic behavior.

Some work on finite computations task models takes into account dependencies between jobs [6] or start times and deadlines [7], but none had dynamic changes of timing behavior (i.e. jobs always have one fixed start time and one fixed deadline).

Finally, some results exist in the literature about scheduling independent of the task model, but they often assume that jobs do not change dynamically [4]. However, these results can be adapted to fit our model, as it is done here to the proof that EDF is optimal [8].

Language interface to time constraints The ΨC language allows to express tasks in this model (Section 4.2) by indicating time constraints using a language interface, rather than using an API (like POSIX). This allows the application designer to focus on his needs rather than on scheduling.

This is similar to synchronous programming languages like Esterel [9] or Lustre [10], or time-triggered programming language such as Giotto [11]. The main diffence of our work with these languages is the underlying task model, which is not restricted to periodic or sporadic tasks, but rather add timing constraints to any automata.

Time-triggered architecture and models The time-triggered architecture [12] focus on separation of a hard real-time system between interfaces and components. The interface consist of asynchronous communication at a priori known sampling dates. The components in this architecture is often periodic or sporadic.

3 Time-constrained tasks

The time-constrained task model allows accurate description of the time constraints of single-threaded time-constrained computations using graphs called *time-constrained automata*. Concurrent time-constrained computations are represented by multiple automata.

We present time-constrained automata in three steps: introduction to the model given for chains and trees before explaining the time-constrained automata.

3.1 Chains

Blocks, arcs, nodes and their relationships A *block* is a sequence of instructions taken as a whole (the term "block" comes from the control flow graph terminology). Blocks are represented by *arcs* and are separated by *nodes*. The node from which the arc starts *immediately precedes* the arc, and the node to which it leads *immediately succeeds* the arc.

From the relations *immediately precedes* and *immediately succeeds*, we can derive by transitive closure the relations *precedes* and *succeeds*. It can be said that an arc succeeds an arc, an arc succeeds a node, a node succeeds an arc, or a node succeeds a node (resp. precedes).

A *chain* is a sequence of blocks, executing one after the other. When blocks *a* and *b* are consecutive, instructions of *a* have to be executed before those of *b*.

Chains have a *first node*, at which they start, but can be *infinite*.

Temporal constraints As a chain is a sequence of indivisible blocks, time constraints can apply only to blocks. Only two kind of constraints are possible:

- A block can be constrained to start only *after* a certain *date* (i.e. specific time instant), or
- it can be constrained to finish *before* one.

As TCA target hard real-time systems, time constraints are strict, i.e. failure to meet them is an incorrect behavior. We choose to make nodes bear the constraints:

- "After constraints" of a block are borne by the immediate predecessor node, denoted by $\triangleright$;
- "Before constraints" of a block are borne by the immediate successor node, denoted by $\triangleleft$;
- When a block bears both a before and an after constraint *at the same date*, then it becomes a *synchronization point*, represented by $\diamondsuit$.
- Except for synchronization points, nodes cannot bear more than one constraint[1].
- Nodes with no constraint are represented by $\bigcirc$.

We label the constraint nodes by the absolute date that they represent. Figure 1 gives an example.

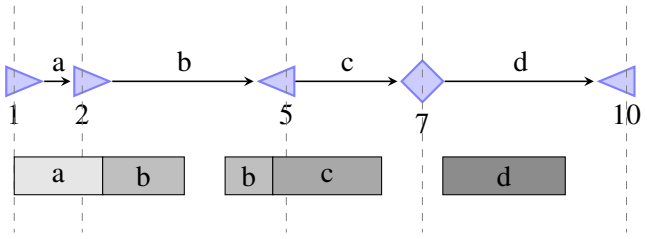


Figure 1: Constrained chain: *a* must start after date 1, *b* must execute between 2 and 5, *c* must end before 7, and *d* must execute between 7 and 10. Beneath is a possible corresponding preemptive schedule.

[1] Allowing more than one constraint per node would not extend the expressive power, because of constraints redundancy seen below.

Extension of constraints to other arcs An *after* node implicitly constrains all the succeeding blocks to start after its date, and a *before* node constrains all the preceding blocks to end before its date. This is formally stated by the following lemma:

Lemma 1 (Implicit extension of constraints to other arcs)**.** *If a block b succeeds an after node A of date d, then b must start after d.*

If a block b precedes a before node B of date d', then b must end before d'.

Proof. The lemma is proved by recurrence:

Either b immediately succeeds A, then by definition b must start after d.

Else assume the lemma is true for the block c at distance n from A, i.e. c must start after date d; then the node at distance $n+1$ from A (if any) immediately succeeds block c, so executes after it, and must start after d.

The proof is similar for the *before* node. □

From the graphical representation, one can easily derive the implicit constraints on a block. For instance in Figure 1, a must end before 5, because the "5" before constraint succeeds a.

Thus time-constrained automata are a model based on *possible intervals of computations*, i.e. each block has a particular interval during which it can execute. By contrast, the synchronous model is based on single *points* of computation.

Impossible and redundant constraints There are also some relationships between the dates of the different constraints: some cannot be satisfied, others can be simplified because they are implied by another one.

Figure 2 shows the different cases where a constraint is followed by a constraint with an earlier date.

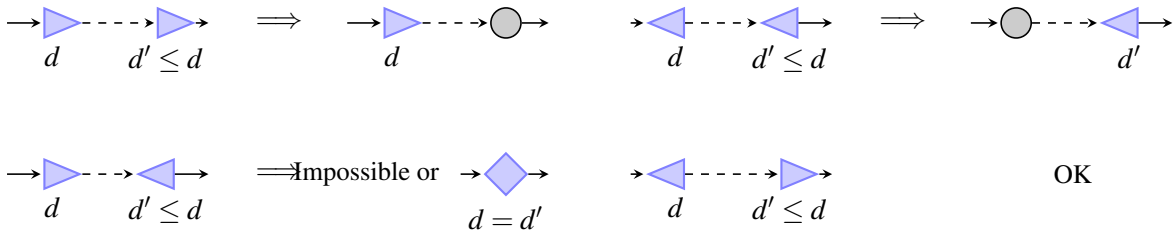


Figure 2: Possible simplifications when a node precedes a node with earlier date.

Theorem 2. *When a node N of date d precedes a node N' of date $d' \leq d$, then*

- *If N and N' are after nodes, then N' can be removed.*
- *If N and N' are before nodes, then N can be removed.*
- *If N is an after node and N' a before node, then either both can be merged into a synchronization point or the constraints are impossible to fulfill.*

Proof.

- If N and N' are both after constraints: then by Lemma 1, the block immediately succeeding N' is already implicitly constrained to start after $d \geq d'$. So the constraint is redundant.
- Similarly, if N and N' are both before constraints, Lemma 1 implies that the N constraint is already implied by N', and is redundant.
- If N is an after constraint and N' a before constraint:
  - If $d' < d$, then any block between N and N' is constrained to start after it ends, which is a condition impossible to fulfill.

– Else $d = d'$: either a block between N and N' is not empty (i.e. has at least one instruction), in which case we must execute some instructions in 0 time, which is impossible. Else all blocks between N and N' are empty. Then we can replace N, N' and the blocks between by a single synchronization node of date d: by Lemma 1, the constraints of all blocks after N' and before N are preserved.

□

Thus, there is only one case where it is useful that a node precedes a node of earlier date, which is the fourth case of Figure 2.

Relative labeling of constraints The following is an immediate corollary of Theorem 2:

Corollary 3. *A chain can be simplified so that the dates of all constraint nodes following an after node of date d are greater than d.*

Thus the following labelling convention can be adopted without modifying the semantics of our model: *all node dates can be labeled using the relative date from the previous after node* (including synchronization points). This convention will allow expression of loops in automata. We denote the use of relative labeling by putting underscores below dates. The first node of a chain is labeled relatively from 0.

Figure 3 represents the re-labeling of Figure 1 with relative dates.

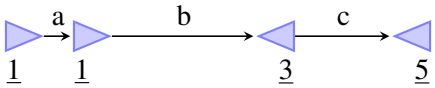


Figure 3: Chain of Figure 1 with relative labeling

Chains can be used to model the history of a job release by a task. However they have some obvious limitations: infinite computations can only be modeled by infinite chains, which cannot be written in a specification; and chains cannot model conditional execution of blocks.

3.2 Time-constrained trees

We extend the previous concept of chains to *trees*, which requires to handle "choices".

We now allow several blocks to start from a node (such a node is called a *choice node*). This expresses the fact that different execution paths may be taken. The choice of which path to take is made when finishing executing the immediately preceding block. Figure 4 gives an example.

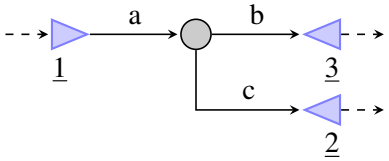


Figure 4: Depending on the execution of a, either b or c will be executed. The path $a \to b$ has 2 units of time to complete, but the path $a \to c$ only has 1.

All paths in the graph constitutes chains, and the "precedes" and "succeeds" relationships are still defined. Thus the previous theorems and lemmas that applied to chains still hold.

3.3 Automata

To represent infinite computations with finite objects, we use automata. Automata differ from trees in two aspects: first they allow several arcs to finish on the same node; second they allow cycles in the graph. These differences change the *precedes* relation on blocks, and affects negatively the semantics, which depends on this relation. We address these problems by defining the semantics of automaton as such: the semantics of a time-constrained automaton is equivalent to that of its *unfolded tree*, which is the infinite tree representing all possible traversals of the automaton. Thus, the *precedes* relation, and the semantics, are preserved.

Moreover, time in automata is expressed using relative labelling, and the unfolding operation also performs the conversion to absolute labelling.

Figure 5 shows example of such an unfolding.

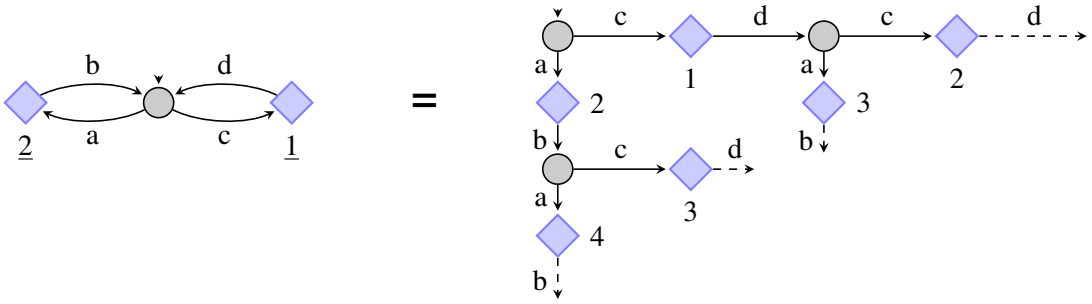


Figure 5: Unfolding of an automaton

This semantics of automata using "unfolding" allows to preserve the "precede" and "succeed" relationships as defined on trees, which are the basis for the semantics of time-constrained computations.

Note that all time-constrained computations cannot be represented by a finite automaton. For instance, an algorithm executing a block in loop, with the k^{th} iteration constrained to execute between instants 2^k and 2^{k+1} (because relative constraints can only be added). This case is of little practical interest though; in fact, we believe that most interesting cases are representable with automata (with only one extension not shown here for the sake of simplicity, which is the synchronization with a different clock).

4 Applications

In this section we present some basic examples of time-constrained automata application, as well as an implementation based on the ΨC programming language.

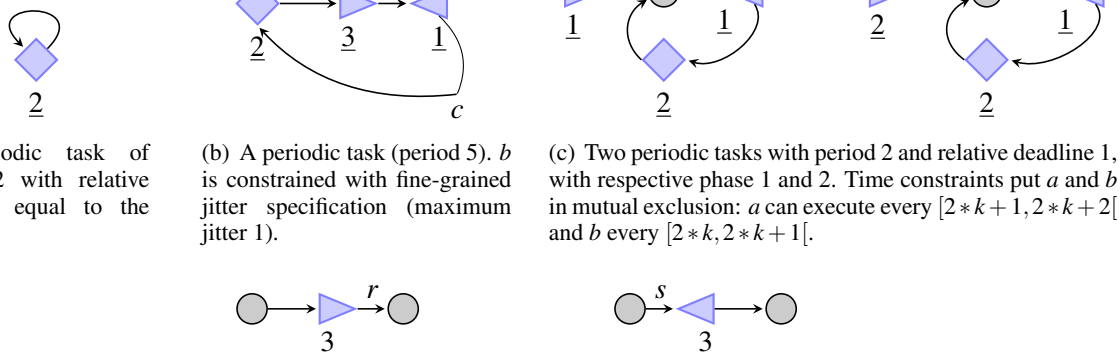
4.1 Example uses

Time-constrained automata can be used to accurately model tasks timing requirements in a great variety of situations, as it is a superset of existing time-triggered models.

The basic example is modeling periodic tasks with deadline equal to their period [13] (Figure 6(a)). More evolved usage are general periodic tasks, whose deadlines can be smaller or even larger than periods (Figure 6(c)). Fine-grained maximum jitter may also be specified (Figure 6(b)).

A periodic task might require an initialization stage, which may have time constraints (this may be the case, e.g., for device initialization). This also allows to phase different periodic tasks, as in Figure 6(c), which allows implementation of mutual exclusion based on time.

The before and after constraints are also well-suited to synchronize tasks that must communicate. Figure 6(d) gives an example of a sender and a receiver; the problem is detailed in section 4.3.



(a) Periodic task of period 2 with relative deadline equal to the period.

(b) A periodic task (period 5). b is constrained with fine-grained jitter specification (maximum jitter 1).

(c) Two periodic tasks with period 2 and relative deadline 1, with respective phase 1 and 2. Time constraints put a and b in mutual exclusion: a can execute every $[2*k+1, 2*k+2[$ and b every $[2*k, 2*k+1[$.

(d) Synchronisation using time: s sends a message before time 3, r receives it after time 3. This guarantees that the message will always be received.

Figure 6: Example uses of time-constrained automata

Other applications include going into a degraded mode, or communicating with device that have complex timing requirements. For instance we easily developed a mouse and keyboard driver in OASIS that work without using hardware interrupts. Different stages (detection of plug/unplug, initialization, normal polling) have different timing requirements, which can be described accurately with time-constrained automata.

To sum up, time-constrained automata are a powerful tool to specify time-constrained tasks that have timing requirements. But they can be more than that: in the next sections, we show how these automata can be derived from source code and be used for scheduling.

4.2 Writing time-constrained programs

ΨC is a programming language designed for implementing time-constrained automata. ΨC preserves the operational semantics of C, but adds time constraints to these semantics with the Ψ extension (this extension could be applied to any imperative programming language).

C control flow graphs are automata, so C's instructions for control flow can be used to express sequencing of blocks, loops, and choices. The basic Ψ addition to C is the addition of `before`, `after`, and `advance` instructions that respectively add before and after constraints, and synchronization points. It then becomes possible to express time-constrained automata in ΨC. There are other extensions to ΨC, to express for instance communication between agents (with automatic buffer sizing) and synchronization between different clocks.

Figure 7 gives a ΨC code excerpt with the corresponding automaton. (Note that our automata differs slightly from usual control flow graphs because code is carried by arcs, instead of nodes.)

4.3 Safe interaction of time-constrained tasks

One of the most interesting aspects of the time-constrained methodology is that it allows to define communication primitives for safe interactions between tasks.

Classical problems of task interactions Common problems in communication and synchronization primitives include:

- Deadlock, happening when synchronizing communications (mutex, synchronous IPC...) happen in the wrong order.

```
while(1) {
  after(1);
  if(...)  {
    after(2);
    a:
    for(i=0;i<10;i++)
      { b:
        advance(1);
        c: }
  d: }
  else before(2);
  e:
  advance(5);
}
```

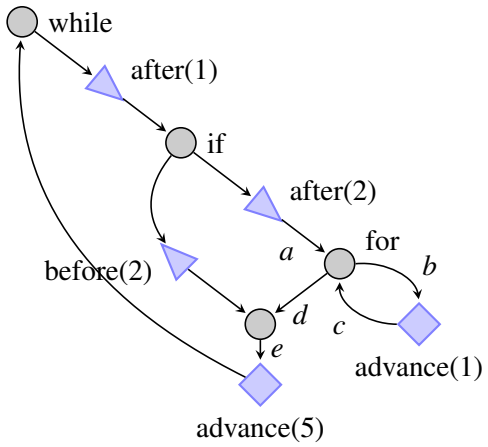


Figure 7: A ΨC code excerpt and corresponding automaton. Blocks *a, b, c, d,* and *e* are labelled in both the code and the automaton. As an example "*a;b*" and "*c;b*" must be executed within 1 unit of time; "*c;d;e*" must be executed within 5.

- Change in scheduling, happening for every synchronizing communication. These makes scheduling and schedulability analysis much more complex [14] (e.g. to take into account priority inheritance).
- Nondeterminism, which can happen for every kind of communication. This makes executions hard or impossible to reproduce, rendering tests useless.
- Insufficient buffer sizes, which happens for buffered communication. This can lead to unpredictable blocking time.

The communication primitives we provide, use time constraints to avoid all these pitfalls:

Synchronization First, we do not provide any synchronization or synchronizing primitive: all communications are asynchronous, but their ordering is controlled by the time constraints. For instance, Figure 6(d) shows how we can enforce block *r* to happen after block *s*, by choosing an arbitrary date (3) for "synchronization". This slightly over-constrains the system; but the benefits of doing so (simple optimal scheduling, simple schedulability analysis) also lead to a gain of performance, and we think this is a good trade-off.

Determinism Second, message communication can be made deterministic. The first source of nondeterminism occurs because of ordering problems between the sender and the receiver. See for instance communication using a variable in shared memory: if the variable is written by the sender before it is read by the receiver, then communication happens; if the order is reversed, it does not happen. Using time constraints, the sending block can be enforced to end before the receiving block begins, using a "synchronization date" as in Figure 6(d). But nothing prevents another block in the receiver to try to read the value before this synchronization date, an operation that nondeterministically succeeds or fails.

This problem is solved by introducing the *visibility date* concept. A communication can be made only if the receiving block is always after the visibility date (i.e. the receiving block succeeds an after node whose date is greater than the visibility date), and if the sending block is always before the visibility date (i.e. it precedes a before node whose date is smaller than the visibility date). This is achieved by tying the communication primitives to the time constraints, and using an appropriate implementation of the communication primitives.

The second source of nondeterminism occurs when multiple agents send a message to another agent at nearly the same time. If the reception depends on sending order, message reception can be nonde-

terministic. This can be solved using appropriate implementation, and is largely independent from the time-constrained model. The combination of these two techniques allows implementation of communication primitives that are provably deterministic.

Buffer sizes The last possible caveat is buffer size. If a buffer is not large enough to contain all the messages, a run-time situation can occur where the sender cannot store the message it wants to send. Then it must either block (but we do not want to provide synchronizing primitives) or throw a runtime error, which is difficult to analyze.

Automatic buffer computation efficiently solves this problem. Time constraints allows to infer the respective rates and phases of communication production/consumption, which allows to know when buffer parts can be re-used, and to infer the exact sizing of the buffer. Details vary according to the communication primitive used.

OASIS primitives for safe interaction Thus time constraints are of great help for designing communication primitives for safe interaction. We have implemented several such primitives: *temporal variable* implements a 1-to-n regular data flow, while *message* are n-to-1 irregular communication. Several others are being implemented in the context of the PharOS project. More details can be found in [2, 15].

In practice, the methodology for designing OASIS applications consists in writing Gantt charts with the timing constraints of the receiver and sender tasks, and of the communication primitives. The periods and phases are tuned to respect the end-to-end requirements of the tasks. But we are currently working on a more formal methodology for designing OASIS applications.

4.4 Experience with the time-constrained methodology

Writing real-time applications using the time-constrained methodology (and ΨC programming language) allows to write safe, parallel programs with ease (see [3] for a large practical example from the nuclear industry). The specification, design and implementation are tightly coupled, which greatly simplifies verification and validation. Verification and validation is generally the most costly phase when designing a real-time system, especially when it is safety-critical.

Parallel programs in ΨC are *deterministic*, i.e. have predictable and reproducible execution. Hence, tests are reproducible despite the parallel execution.

Time-constrained tasks synchronize only using time, and do not use mutexes or semaphores. This allows us to perform *exact* feasibility analysis, and to reach high processor utilization, even on multiprocessors. This also provides safety guarantees (deadlock is impossible).

5 Scheduling of time-constrained tasks

This section presents the precise scheduling semantics of the time-constrained automata, and gives an optimal scheduling algorithm on single processor based on EDF. We decompose again the presentation in three parts: scheduling of chains, trees, and automata.

5.1 Definitions

Required execution time function We assume the existence of a function $||\cdot|| : \mathit{Blocks} \to \mathbb{R}$ that gives the execution time necessary to complete a block. Note that this function is not necessary to define the semantics of TCA.

Validity and correctness We say that a schedule is *valid* when it respects the semantics of the task model (e.g. executes a job only between its start time and deadline). We say that it is *correct* when it is valid and tasks have enough time to complete before deadline.

Feasibility and optimal algorithm A set of tasks is *feasible* if there exists a correct schedule. An *optimal* scheduling algorithm finds a correct schedule whenever one exists (i.e. whenever the set of tasks is feasible).

5.2 Scheduling of time-constrained chains

5.2.1 Semantics

Schedule mapping On single processor computers, a schedule is a function $s : Blocks \times \mathbb{R}_+ \to \{0,1\}$ that tells whether block b is scheduled at time t.

In an interval of time $[t_1,t_2]$, a block b is executed for a duration of $\int_{t_1}^{t_2} s(b,t)\mathrm{d}t$

For multiprocessor systems, the definition is the same, because the specific placement on the processors can be abstracted [16].

Conditions for a valid schedule of time-constrained chains Conditions for a function s to be a valid schedule express the sequentiality of blocks and the time constraints in the schedule:

- If a block b has an "after" constraint of date d, it must not be scheduled before d:

$$\forall t, \quad t < d \;\Rightarrow\; s(b,t) = 0$$

- If a block b has a "before" constraint of date d', it must not be scheduled after d:

$$\forall t, \quad t > d' \;\Rightarrow\; s(b,t) = 0$$

- If block b precedes block b', it must be scheduled before b': (note: the two formulas are equivalent)

$$\forall t', \quad \big(s(b',t') \neq 0 \;\Rightarrow\; \forall t > t',\, s(b,t) = 0\big) \quad\Longleftrightarrow\quad \forall t, \quad \big(s(b,t) \neq 0 \;\Rightarrow\; \forall t' < t,\, s(b',t') = 0\big)$$

Condition for a correct schedule A correct schedule is a valid schedule, with the condition that all blocks b must be scheduled for at least their required execution time:

$$\int_{t=0}^{+\infty} s(b,t)\mathrm{d}t \;\geq\; ||b||$$

Note the use of $\geq$ instead of $=$ in the previous equation: this allows feasibility analysis to allocate more CPU time than necessary.

5.2.2 Optimal scheduling algorithm on single processors

Current real-time scheduling algorithms consider tasks as releasing a sequence of static jobs [4]. This is not applicable to our model because a time-constrained task has "changing jobs", i.e. job's execution time and deadline can change dynamically.

We propose EDF-dyn, an extension to EDF to schedule a set of time-constrained chains:

Definition 4 (EDF-dyn)**.** EDF-dyn schedules at each instant t, the task whose current block's implicit deadline is the soonest, chosen among all tasks whose current block's implicit start date is sooner than the current date. Ties can be broken arbitrarily.

For a set of chains $\mathscr{C}$, we note `EDF-dyn`$(\mathscr{C})$ the schedule produced by EDF-dyn.

The proof that EDF-dyn is optimal on time-constrained chains can be readily adapted from the proof that EDF is optimal on static jobs:

Theorem 5. *EDF-dyn is optimal for time-constrained chains on single processors : if $\mathscr{C}$ is feasible, then* $\texttt{EDF} - \texttt{dyn}(\mathscr{C})$ *is a correct schedule.*

Proof sketch. The proof is the same as the standard EDF optimality proof (e.g. [4]), obtained by replacing "job" by "block" and "deadline" by "implicit deadline". The proof is by absurd: if EDF-dyn is not optimal, and t_0 is the last instant where no more correct schedule agrees with EDF-dyn, we construct a correct schedule that agrees with EDF-dyn until $t_o + \delta$. □

EDF-dyn can be efficiently implemented, as seen in the next section.

5.2.3 Implementation

High-level algorithm

- When a scheduling decision is made, the algorithm selects the task on the ready list with the smallest current deadline;
- The scheduler is awaken only when:
  1. the current date becomes the same as the date of some "after" node; the current task is added to the ready list, and a new decision is made;
  2. an "after" node is reached, whose date is bigger than the current date; the current task is removed from the ready list, and a decision is made;
  3. a "before node" is reached. The "current deadline" of the task is changed, and a decision is made.

Intuitively, case 1. represent new job becoming available; case 2. the fact that a task has to wait before continuing execution; case 3. that execution of a blocks finishes before its deadline.

Note: This algorithm behaves as standard EDF on regular jobs.

Theorem 6. *This algorithm implements EDF-dyn.*

Proof sketch. The instants where the scheduler is awaken are sufficient to make the ready list and "current deadline" of the tasks always up-to-date. □

Implementation in OASIS EDF-dyn is implemented in the OASIS kernel[2]. To wake the scheduler when "after" and "before" nodes are reached (corresponding to `after` and `before` Ψ instructions in source code), these are replaced by system calls at compilation.

The chains are simplified using Theorem 2, and "before" nodes carry information about the deadline of the following "before" node, so that the next deadline is known without complex lookup. Thus, "before" nodes are transformed into "update deadline" system calls.

Finally, we also check that deadlines are not missed by waking the scheduler up at deadline dates; different blocks can be taken upon deadline miss (from system stop to degraded mode). We can also check that a block b is not executed more than $||b||$.

5.3 Scheduling of time-constrained trees

5.3.1 Semantics

From trees to chains A path in a time-constrained tree is a time-constrained chain. Given a set of trees $\mathscr{T}$ and a set of choices in the tree $\mathscr{U}$, we define the operator $\mathscr{E}$ so that $\mathscr{E}(\mathscr{U}, \mathscr{T}) = \mathscr{C}$ is the set of time-constrained chains obtained by extracting from the trees only the paths corresponding to the choices in $\mathscr{U}$.

Tree schedule The schedule for a tree has to take into account the fact that depending on the choices made, multiple time constraints are possible; so a schedule for a tree is a function of the choices made.

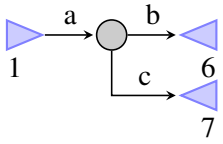
We refer to the instant when the block preceding the choice node has finished executing to the *instant of choice*. As the choice is not known by the scheduling algorithm until this instant, the schedules must be the same up to it.

More formally, we define a valid schedule for a set of trees $\mathscr{T}$ to be a function S as such:

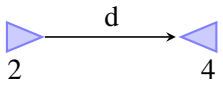
- S associates to each possible set of choices $\mathscr{U}$, $S_{\mathscr{U}}$ which is a valid schedule mapping for the set of chains $\mathscr{E}(\mathscr{U},\mathscr{T})$;
- Additionally, for two different sets of choices $\mathscr{U}$ and $\mathscr{U}'$, $S_{\mathscr{U}}$ and $S_{\mathscr{U}}$ must coincide up to the time τ when the first choice node that differentiates them is reached:

$$\forall t < \tau,\ \forall b, \quad S_{\mathscr{U}}(b,t) = S_{\mathscr{U}'}(b,t)$$

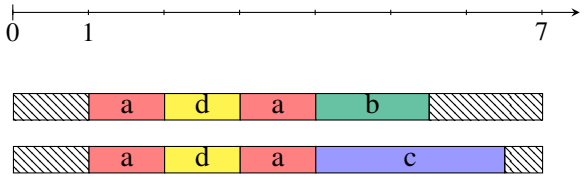
Figure 8 gives an example.



(a) Task 1



(b) Task 2



(c) Valid schedule

(d) Invalid schedule : differ before the instant of choice

Figure 8: Here, $||a|| = 2$, $||b|| = 2$, $||c|| = 1$ and $||d|| = 1$; there is one choice so a tree schedule is two schedule mapping. Instant of choice (i.e. last time a is executed) is at time 4.

Correctness conditions We define a schedule S to be correct for a set of trees $\mathscr{T}$ when all schedules on the corresponding chains are correct:

$$\forall \mathscr{U}, \quad S_{\mathscr{U}} \text{ is correct.}$$

In Figure 8, the schedule 8(c) is correct because if block c is chosen, the first schedule mapping is correct for the chains $a \to c$ and d; and if block b is chosen, the second is correct for the chains $a \to b$ and d.

This means that all possible executions can be correctly scheduled.

5.3.2 Choice deadline inheritance

Online, deadline-based algorithms such as EDF always need to know the next deadline; when scheduling a tree, this "next deadline" is not known because it depends on future execution. Choice deadline inheritance solves this problem.

We define *choice deadline inheritance* as follows : for each "choice" node, we consider the (temporally) closest "before" node in all possible following nodes. If τ is the date of this "before" node, we add to the choice node a "before" constraint of date τ.

Note: If there are no following "before" nodes, we do nothing. This can be viewed as inheriting a deadline at date $+\infty$.

On a set of trees $\mathscr{T}$, we denote by $\texttt{CDI}(\mathscr{T})$ the set of trees transformed with choice deadline inheritance. Figure 9 provides an example.

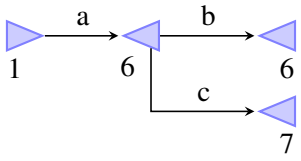


Figure 9: Figure 8(a) transformed with CDI. The smallest deadline is 6, so the choice node "inherits" this deadline.

The following theorem shows that choice deadline inheritance, which adds constraints to the time-constrained trees, does not change its schedulability:

Theorem 7. *A schedule is correct for a set of time-constrained trees $\mathscr{T}$ iff it is correct for $\texttt{CDI}(\mathscr{T})$.*

Proof. As CDI add constraints, any schedule correct for $\texttt{CDI}(\mathscr{T})$ is correct for $\mathscr{T}$.

If a schedule is correct for $\mathscr{T}$, we consider a path of the tree choosing the soonest deadline. For this path, the block before the choice node is implicitly constrained to finish before this deadline. And because of the validity condition on trees, all schedules for the tree must be the same until this choice node is reached, so they all finish before this deadline. Thus the additional constraint expressed by $\texttt{CDI}(\mathscr{T})$ is fulfilled. □

This proves that the constraints added by choice deadline inheritance do not affect feasibility. We will now see how they are used to implement EDF scheduling on trees.

5.3.3 Optimal scheduling on single processor

We now define the EDF-dyn-min algorithm (EDF-dyn with minimal deadline) algorithm on time-constrained trees:

Definition 8. EDF-dyn-min schedules at each instant t, the task whose current block's *possible* implicit deadline is the soonest, chosen among all tasks whose current block's implicit start date is sooner than the current date. Ties can be broken arbitrarily.

The schedule of Figure 8(c) is the schedule obtained with EDF-dyn-min.

Lemma 9 is central to the online implementation of EDF-dyn-min:

Lemma 9. *Let $\mathscr{T}$ be a set of trees, and $\mathscr{U}$ a set of choices for these trees. Then*

$$\texttt{EDF-dyn-min}(\mathscr{T})_{\mathscr{U}} \;=\; \texttt{EDF-dyn}(\mathscr{E}(\mathscr{U}, \texttt{CDI}(\mathscr{T})))$$

That is, for a given set of choices $\mathscr{U}$, the schedule of EDF-dyn-min can be obtained by executing EDF-dyn *on the paths of $\texttt{CDI}(\mathscr{T})$ corresponding to $\mathscr{U}$.*

Proof sketch. The soonest possible implicit deadline in the definition of EDF-dyn-min is the constraint's date added by the choice deadline inheritance algorithm. Thus for each possible path, EDF-dyn is followed with this additional constraint on choices. □

Theorem 10. *EDF-dyn-min is optimal on time-constrained trees.*

Proof sketch. Let $\mathscr{T}$ be a feasible set of time-constrained trees and $S = \texttt{EDF-dyn-min}(\mathscr{T})$ be the schedule produced by EDF-dyn-min.

For any set of choices $\mathscr{U}$, by Lemma 9 and because EDF-dyn is optimal, $S_{\mathscr{U}}$ is a correct schedule for $\mathscr{E}(\mathscr{U}, \mathscr{T})$.

It remains to prove that for two choices $\mathscr{U}$ and $\mathscr{U}'$, the schedules remain the same until the first instant of choice. Following Lemma 9 and definition of choice deadline inheritance, all deadlines are the same until the first differing choice is taken; thus the schedules are the same until this moment. □

Implementation in OASIS The implementation of EDF-dyn-min in OASIS executes only trees with choice deadline inheritance. By Lemma 9, it uses the EDF-dyn algorithm given in Section 5.2.3.

To implement choice and choice deadline inheritance, the compiler adds two separate "update" system calls, at the beginning of each "then" and "else" branches of each choice that changes timing behavior.

In fact, the "update" system call for the branch with the soonest deadline is redundant and can be removed.

5.4 Scheduling of time-constrained automata

The semantics of scheduling a set of time-constrained automata $\mathscr{A}$ is the semantics of scheduling the corresponding set of unfolded trees.

Implementation in OASIS We use the same algorithm as for time-constrained trees: we apply choice deadline inheritance to the automaton and use the EDF-dyn algorithm.

By replacing `after` instructions by after system calls, and `before` instructions by "update deadline" system calls, the time-constrained automaton is completely embedded in the code structure. Execution of this code unfolds the automaton on the fly. Conversion from relative to absolute labelling is also performed dynamically.

Feasibility analysis Although we won't present it due to lack of space, it is possible to conduct a feasibility analysis on time-constrained tasks. This is done by computing the product of the automata to analyze the blocks simultaneously executed, and translate this product into a linear programming (in fact, network flow) problem (variables represent the amount of a block done on an interval). Although of great complexity in the worst case, this proved to be very tractable for all the industrial problems we have considered.

One of the reason why the problem is tractable is the absence of interaction between communication and scheduling. These interactions are often hard to take into account in practice.

6 Conclusion

The time-constrained task model is a general model able to accurately describe the temporal behavior of algorithms constrained by time. One of its main interest is that it can be used for scheduling hard real-time tasks, with high efficiency since optimal scheduling exists. We have presented their particular scheduling semantics, and some of their uses (guaranteed safe interaction, deterministic communication, and derivation from source code).

We are currently writing a complete formalization of the scheduling semantics, as well as very detailed proof of the optimality of our scheduling and feasibility analysis algorithms. This will allow to provide formal, abstract semantics for the time-constrained automata and the different communication primitives, for use by future model-checking tools.

A forthcoming paper will present feasibility analysis in detail, as well as results on multiprocessor scheduling of our model. Indeed, our non-blocking model of tasks can be executed with very high utilization, even on multiprocessor computers, which makes it very promising for high-performance real-time systems. Even if "the future" needs to be known for optimal scheduling [17], the time-constrained model only expresses a reasonable set of possible futures, thus allowing near-optimal scheduling on multiple processors.

Present and future extensions The model presented here is sufficient to study schedulability, but some extensions make it more practical to build real applications using a time-constrained model. Among these are communication primitives, synchronization on multiple clocks (that allow a task to synchronize with another one that has a different rate), variable afters (the date is only known to be in some interval)... All these extensions are implemented in the industrial version of OASIS.

A future extension of particular interest with regard to scheduling is the extension of the model to allow multithreaded computations. This is done by allowing certain nodes to "create" new automata (a `fork()`-like node), and some other to wait for their completion (a `wait()`-like node). Another future extension is support for event-triggered computation in the model.

A last research direction is the study of automatic transformation from formal specifications (e.g. timed automata [1]) to our task model.

References

[1] R. Alur and D. Dill, "A theory of timed automata," *Journal of theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[2] D. Chabrol, V. David, C. Aussaguès, S. Louise, and F. Daumas, "Deterministic distributed safety-critical real-time systems within the OASIS approach," in *17th IASTED PDCS'05*, November 2005.

[3] V. David, C. Aussaguès, S. Louise, P. Hilsenkopf, B. Ortolo, and C. Hessler, "The OASIS based qualified display system," in *Proceedings ANS (NPIC&HMIT 2004)*, 2004.

[4] S. Baruah and J. Goossens, "Scheduling real-time tasks: Algorithms and complexity (j.y.t leung eds): Handbook of scheduling: Algorithms, models and performance evaluation. chapman et al," 2004.

[5] S. K. Baruah, "A general model for recurring real-time tasks," in *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, (Washington, DC, USA), p. 114, IEEE Computer Society, 1998.

[6] R. R. Muntz and J. E. G. Coffman, "Preemptive scheduling of real-time tasks on multiprocessor systems," *J. ACM*, vol. 17, no. 2, pp. 324–338, 1970.

[7] W. A. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, pp. 177–185, 1974.

[8] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," in *IFIP Congress*, pp. 807–813, 1974.

[9] G. Berry, *The Foundations of Esterel*. MIT Press, 2000.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.

[11] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Lecture Notes in Computer Science*, vol. 2211, pp. 166+, 2001.

[12] H. Kopetz, "The time-triggered model of computation," *Real-Time Systems Symposium, IEEE International*, vol. 0, p. 168, 1998.

[13] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, 1973.

[14] M. Lemerre, V. David, and G. Vidal-Naquet, "A communication mechanism for resource isolation," in *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pp. 1–6, ACM, 2009.

[15] C. Aussaguès, D. Chabrol, V. David, D. Roux, N. Willey, and A. Tournadre, "An os for multicore embedded systems compliant with automotive safety standards," in *Proceedings of the International Automotive Electronics Congress, Embedded systems and Standardisation Session, Paris France*, November 2009.

[16] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet, "Equivalence between schedule representations: Theory and applications," *RTAS*, vol. 0, pp. 237–247, 2008.

[17] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.